Noname manuscript No.
(will be inserted by the editor)

Optimization of Inland Shipping

A Polynomial Time Algorithm for the Single Ship Single Lock Optimization Problem

Jens Hermans

the date of receipt and acceptance should be inserted later

Abstract In this paper we explore problems and algorithms related to the optimization of locks, as used in inland shipping. We define several optimization problems associated with inland shipping. We prove that the problem of scheduling a lock is NP-hard if one allows multiple ships to go through in the same lock operation. The single ship lock optimization problem can however be solved in polynomial time and a novel deterministic scheduling algorithm for solving this problem is presented in this paper.

1 Introduction

Inland shipping is faced with some interesting challenges in modelling and optimizing traffic on waterways. Multiple locks, which are used to connect different water levels, and movable bridges on canals have to be operated, which creates the opportunity to make a detailed schedule and reduce waiting times for ships at these locks and bridges. In ideal circumstances it should be possible to eliminate all waiting times, creating a 'green' wave, like it is known in road traffic, where traffic lights are synchronized.

There are however some important differences with road traffic. First, given the limited number of ships on a waterway, it is impossible to model them in groups. Every ship must be modelled individually. Second, the behaviour of locks does not have a counterpart in road traffic. A lock always has to alternate between up-river and down-river lockings. In some cases it is necessary to do an empty locking (without ships in the lock chamber) to ensure that the lock is in the right starting position for the next ship. A lock also has a fixed capacity: the surface of the lock chamber is limited.

In the cases that are considered in this paper, we assume a relatively small waterway, where every locking will have more or less the same operation time. The time for entering and exiting the locks is negligible or constant, in which case it can be included in the locking time.

1.1 Previous Proposals

Only a limited number of solutions for solving inland shipping optimization problems have been proposed.

A solution based on simulated annealing was implemented by Zhang et al (2007) for optimizing the locks at the Three Gorges Dam and the downriver Gezhouba Dam. There are two unidirectional locks at the Three Gorges Dam and three bidirectional locks at the Gezhouba Dam. With a hybrid algorithm, based upon simulated annealing, a Mixed Integer Non-Linear Program (MINLP) is solved. A clear disadvantage of this approach are the long computation times.

Another approach are heuristic planning algorithms for scheduling multiple locks in series, like for example the algorithm by Ting and Schonfeld (1998). For a single lock the SPF-algorithm (Shortest Processing time First) gives better results than FCFS (First-Come First-Serve). Using an algorithm based on SPF for multiple locks in series, the total waiting time can be reduced further. The situation described here is, however, different from the waterways we consider: the processing times are allowed to be different because tows consisting of multiple barges are considered. On smaller waterways, for which we try to optimize, such long tows do not exist and the processing times are always the

J.Hermans
KU Leuven - ESAT/COSIC and iMinds,
E-mail: jens.hermans@esat.kuleuven.be

same. Since the whole algorithm is based on the differences in processing times, it cannot be used.

There is also an algorithm based on MINLP for scheduling a single lock by Mundy and Campbell (2005). But, as in the previous case, this situation also assumes tows consisting of multiple barges.

In Hermans (2008) a practical solution is presented for a single lock using Mixed Integer Linear Programming (MILP), together with a theoretical approach which was used as a basis for this paper.

Commercial software solutions for planning of inland shipping, including all the necessary communications, visualisation... , have been developed (e.g. FKS (2007)). These solutions, however, do not explicitly optimize the traffic: they only provide simulations of the traffic.

1.2 Problem Definition and Notation

We limit ourselves to a single lock with unit processing times. In this case the goal is to minimize the waiting times of the ships at the lock. We consider n ships T_i, $i \in [1, n]$, each with an arrival time r_i, width w_i, length l_i and a travelling direction $o_i \in \{A, B\}$. The lock has a locking time, that is assumed to be 1 (all times can always be rescaled to guarantee the locking time is indeed 1), a width w_{lock} and length l_{lock}. The optimization algorithm should assign each ship T_i a starting time s_i. Throughout this paper, subscript indices are often used without explicitly mentioning the ranges for the subscript. In this case the subscripts are assumed to be in the range $[1, n]$. A closed interval (including endpoints) is denoted as $[a, b]$, an open interval as (a, b). Intervals (a, b) with $a \geq b$ are considered equivalent to the empty interval.

The single lock optimization problem, with capacity limitation is an extension of unit-length job scheduling on a single machine with arbitrary release times. Additional constraints ensuring alternation between directions of ships have to be added and also the notion of *empty* operations of the machine has to be added.

Definition 1 *(Single Lock Scheduling Problem - SLSP)* Given a set $\mathcal{T}$ of n tasks (ships) T_i with for each task:

- a release time $r_i \in \mathbb{R}$ and deadline $d_i \in \mathbb{R}$,
- width w_i and length l_i,
- direction $o_i \in \{A, B\}$

and a lock with width w_{lock} and length l_{lock}.

The SLSP is the problem of finding a schedule S that assigns every task a starting time s_i such that

- $r_i \leq s_i, s_i + 1 \leq d_i$ (release time / deadline constraints)
- $\forall s_i, s_j, i \neq j : s_i \notin (s_j, s_j + 1)$ (the tasks either coincide completely, $s_i = s_j$, or do not overlap $s_i \notin [s_j, s_j + 1)$)
- $\forall s_i, s_j, o_i = o_j, s_i \neq s_j : s_i \notin (s_j - 2, s_j + 2)$ (alternating directions)
- $\forall t, \mathcal{S} \subseteq \mathcal{T}, \forall T_i \in \mathcal{S}, s_i = t : Fit(\{(l_i, w_i)\}_{i:T_i \in S}) = true$. Where $Fit(\cdot)$ computes if all ships fit in the lock area or not.

Definition 2 *(Single lock optimization problem with minimax objectif function - SLOPminimax)*

Analogous to definition 1, but the ships do not have a deadline d_i. The SLOPminimax problem is to find a schedule satisfying the same constraints as in definition 1 and as goal function $\min(\max_i(s_i - r_i))$.

An analogous problem *SLOPavgtime* can be defined with goal function $\min(\sum_i(s_i - r_i))$

It is shown in Section 3 that the single lock optimization problem is NP-hard. However, when limiting the capacity of the lock to exactly 1 ship a deterministic algorithm is constructed that creates a schedule in polynomial time. This polynomial time algorithm can be used for scheduling unit length tasks on a single machine with arbitrary release times and deadlines and with additional *class* alternation constraints. A class is equivalent to the travelling direction of the ships and is a property of the tasks. In the case of *class alternation* the machine is imposed the additional constraint that it must alternate between the two classes of tasks. To allow the alternation to continue, even when only tasks of a single class are released, the machine can do *empty* operations.

1.3 Structure of the Paper

In Section 2.1 we discuss several algorithms for the scheduling of unit-length tasks. Section 2.2 introduces the original backscheduling algorithm for unit-length jobs with arbitrary release times and deadlines. In Section 3 an NP-hardness proof of the single lock scheduling problem is presented. Section 4 explains our proposed algorithm in detail. Finally, Section 5 puts forward some challenges for the future.

2 Preliminaries

2.1 Unit-length Job Scheduling

Throughout this section the $\alpha|\beta|\gamma$-notation of Graham et al (1979) is used.

One of the first polynomial time algorithms for single machine scheduling was given by Lawler (1973), in

which he solves $1|\text{prec}|max(f_i(C_i))$ with $f_i(\cdot)$ a non-decreasing function of the completion times C_i and prec some precedence-constraints. In this case, no release times were taken into account and the algorithm can handle arbitrary job lengths.

In general scheduling with arbitrary release times and deadlines is NP-hard, e.g. $1|r_i|\sum w_iTa_i$ (with $Ta_i = max(0, C_i - d_i)$), $1|r_i|\sum Ta_i$ and $1|r_i|\sum C_i$ were shown to be NP-hard (see Brucker and Knust (2009) for an overview of other complexity results for similar scheduling problems).

However, for the case of unit-length job scheduling with arbitrary release times and deadlines, Baptiste (1999) constructed an algorithm for $1|p_i = p, r_i|\sum w_iU_i$ based on dynamic programming. Baptiste (2000) later extended this to a more general goal function $\sum f_j(C_j)$ (under certain conditions for f_j). The high time complexity ($O(n^7)$) and memory complexity ($O(n^4)$) make the algorithm hard to use in practice. Chrobak et al (2006) modified this algorithm for the unweighted case $1|p_i = p, r_i|\sum U_i$ and obtained an algorithm with time complexity $O(n^5)$.

Garey et al (1981) describe an algorithm to solve the problem with arbitrary release times and deadlines in $O(n \log n)$. This algorithm finds a valid schedule (if possible) and minimizes the makespan $\max(C_i)$. This algorithm is very closely related to $1|p_i = p, r_i|L_{max}$ ($L_{max} = \max_i(C_i - d_i)$), in which the maximum lateness is minimized. By using the algorithm from Garey et al (1981) it is easy to solve $1|p_i = p, r_i|L_{max}$, by setting the deadlines to $d'_i = d_i + \Delta t$ and using a bisection search to minimize Δt.

The forbidden zone algorithm by Garey et al (1981) was the best candidate for extension to the single lock optimization problem, even though it has to be embedded in a bisection search.

2.2 The Forbidden Zone Algorithm

Consider n tasks T_i with release time r_i, deadline d_i and unit length. Every task T_i should be assigned a starting time s_i such that $r_i \leq s_i$ and $s_i + 1 \leq d_i$ for all i and in which tasks do not overlap ($\forall s_i, s_j, i \neq j : s_i \notin [s_j, s_j + 1)$). This is a feasibility problem. A schedule that matches these criteria is called a valid schedule.

The algorithm by Garey et al (1981) uses the concept of forbidden zones to solve this problem. A forbidden zone F is an open interval $(\inf(F), \sup(F))$ in which no task is allowed to start if the algorithm produces a valid schedule. In the first phase of the algorithm (backscheduling) the forbidden zones are defined. We use $\mathcal{F}$ to denote the set of all forbidden zones and $\bigcup\mathcal{F}$ to denote the union of all forbidden zones in $\mathcal{F}$. In the second phase (forward scheduling) a valid schedule is created, in which no task starts in a forbidden zone. After the backscheduling phase it is already clear if the algorithm will yield a valid schedule: if backscheduling fails, a valid schedule does not exist. Otherwise, if backscheduling succeeds and gives a list of forbidden zones, the forward phase will always succeed.

2.2.1 Backscheduling

The key of the backscheduling algorithm is a subroutine which takes as input r and d and a set of existing forbidden zones $\mathcal{F}$. The subroutine considers all k tasks T_i with $r \leq r_i < d_i \leq d$. The subroutine will compute the latest time c (also called the 'critical' time), such that there is a schedule of the k tasks between c and d, with no starting time in $\bigcup\mathcal{F}$. The individual release times r_i and deadlines d_i are not taken into account in the subroutine.

So, by adding tasks to the schedule from back to front, a schedule can be easily found. The last task (of the k tasks) starts at $s = d - 1$, unless there is a forbidden zone $F \in \mathcal{F}$ such that $d - 1 \in F$. In this case the task starts at $s = \inf(F) < d - 1$. The next task starts at $s' = s - 1$ or if $s - 1$ is in a forbidden zone at the infimum of the forbidden zone. This process is continued until the last task is scheduled to start at time c. This time is also called the critical time.

In Garey et al (1981) it is shown that, for all alternative schedules in which all k tasks start at a time strictly larger than c (in which no task is allowed to start in a forbidden zone), at least one of the tasks will not be finished before the deadline d. The critical time is, as such, the latest time at which the first among the k tasks must be started to be able to schedule all these tasks.

Example 1 Figure 1 shows an example with 4 tasks and 2 forbidden zones. Task T_4 cannot start at $d - 1$ since $d - 1 \in F_2$ and thus has to start at $s_4 = \inf(F_2)$. T_3 cannot start at $s_4 - 1$ since $s_4 - 1 \in F_1$. For T_2 and T_1 there is no problem and these can start at respectively $s_2 = s_3 - 1$ and $s_1 = s_2 - 1$. The critical time $c = s_1$.

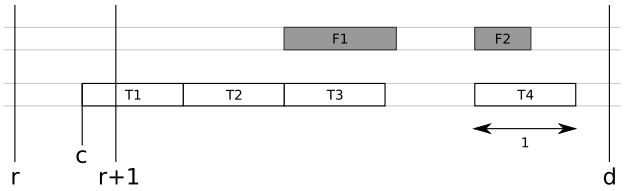

Fig. 1 Backscheduling applied to 4 tasks and 2 forbidden zones.

We distinguish three different situations for the critical time c:

1. $c < r$: in this case the backscheduling algorithm fails, which implies that no valid schedule exists. Since $c < r$, this implies that we should start the first task before r to obtain a valid schedule for the k tasks. This is a contradiction.
2. $r \leq c < r + 1$: in this case $(c - 1, r)$ must be a forbidden zone. If a task (not part of the k tasks we consider) would start in $(c - 1, r)$ this task will be finished in $(c, r+1)$. Since the first of k tasks should start the latest at time c, this gives a conflict with the task ending in $(c, r + 1)$.
3. $c \geq r + 1$: in this case no forbidden zone is defined.

Based on this simple process all forbidden zones can be found by applying it for all r_i and d_j. Care must be taken to ensure that when applying backscheduling all forbidden zones inside the considered interval (r_i, d_j) are already known. So, the release times need to be ordered from high to low when applying backscheduling. For each r_i every possible d_j is used. The critical time obtained when backscheduling with r_i and d_j is called $c_{i,j}$.

For each r_i the smallest critical time $c_i = \min_j c_{i,j}$ is determined. Based on this critical time the forbidden zones are defined as stated in the three cases above.

2.2.2 Forward Scheduling

Once the set $\mathcal{F}$ of all forbidden zones is defined, the forward scheduling algorithm can be executed. Starting with $t = 0$ and repeating the following steps until there are no unscheduled tasks left:

1. If no task is available at time t, set $t \leftarrow \min(r_i : T_i \text{ is unscheduled})$.
2. If t is in a forbidden zone $F \in \mathcal{F}$, set $t \leftarrow \sup(F)$.
3. Choose the task T_i with the earliest deadline from all available, unscheduled tasks. Schedule this task.
4. $t \leftarrow t + 1$.

In Garey et al (1981) it is proven that the forward scheduling algorithm delivers a valid schedule if and only if the backscheduling algorithm does not fail.

3 NP-hardness

Theorem 1 *The single lock scheduling problem is NP-hard.*

The proof is trivial since the lock scheduling problem contains a rectangle packing problem (as described in Lodi et al (2002)), which is NP-hard.

Intuitively, we can say that filling a lock with ships is causing the NP-hardness of the scheduling problem. In reality however, the number of ships in a lock is limited and a obvious question is what would happen if the filling of the lock is ignored (e.g. not taken into account when determining computational complexity) or solved using heuristics.

In the next section an algorithm is derived for scheduling a single lock, when limiting the capacity of the lock to exactly 1 ship.

4 An algorithm for the Single Ship, Single Lock Scheduling Problem

4.1 Problem Definition

Definition 3 *(Single Ship, Single Lock Scheduling Problem with Single Ship Constraint - SLSP1)*

The SLSP1 problem is the SLSP problem with restricted to instances with $w_i = w_{\text{lock}}$ and $l_i = l_{\text{lock}}$ for all T_i.

Definition 3 limits the capacity of the lock to one ship. This also implies that the interval $(s_j, s_j + 1)$ from the non-overlapping restriction in definition 1 can be replaced by a half-open interval $[s_j, s_j + 1)$.

Throughout this section 'empty lockings' (also called 'empty tasks') are used: we allow the insertion of extra tasks in the schedule that are not associated with any task $T_i \in \mathcal{T}$. The empty tasks have a starting time $s_{L,j}$ and have unit length. This allows us to manipulate the empty tasks explicitly, instead of using an implicit definition like the 'alternating directions' restriction from definition 3, which just assures there is enough room to insert an empty task. Empty tasks and their associated starting times are always denoted with the letter L in sub- or superscript.

4.2 General Concepts

Two types of forbidden zones are defined: A-types (denoted as FA), in which no tasks with $o_i = A$ are allowed to start and B-types (denoted as FB), in which no tasks with $o_i = B$ can start. The backscheduling algorithm is extended for two types of forbidden zones.

Consider backscheduling is applied to the interval $[r, d]$, i.e. to all tasks T_k with $r \leq r_k < d_k \leq d$. Let $\#A$ denote the number of tasks among T_k with $o_k = A$ and $\#B$ the number of tasks with $o_k = B$. Two backschedules are constructed: one ending on an A-task and one ending on a B-task. The backschedules consist of an alternating sequence of A and B tasks. If there

are not enough tasks available in a certain direction, extra, empty tasks are inserted.

Algorithm 1 Backscheduling sub-procedure

Require: an interval $[r, d]$, a subset of tasks $\mathcal{S} = \{T_k\}$ with $r \leq r_k < d_k \leq d$ and two sets of previously defined forbidden zones $\mathcal{F}_A$ and $\mathcal{F}_B$

1: **for** $X_{\text{init}} \in \{A, B\}$ **do**
2: $t \leftarrow d$, $X \leftarrow X_{\text{init}}$, $n_A \leftarrow \#\{T_k \in \mathcal{S} : o_k = A\}$ and $n_B \leftarrow \#\{T_k \in \mathcal{S} : o_k = B\}$.
3: **while** $n_A > 0$ or $n_B > 0$ **do**
4: $n_X \leftarrow n_X - 1$.
5: $t \leftarrow t - 1$
6: **if** $\exists F_k \in \mathcal{F}_X$ s.t. $t \in F_k$ **then**
7: set $t = \inf(F_k)$.
8: **end if**
9: $X \leftarrow \bar{X}$.
10: **end while**
11: $c^{X_{\text{init}}} \leftarrow t$, $o^{X_{\text{init}}} \leftarrow \bar{X}$.
12: **end for**
13: Return c^A, o^A, c^B, o^B

Algorithm 1 shows the procedure for backscheduling on (r, d). It returns two critical times c^A and c^B for the backschedule ending on an A-task and a B-task respectively. We will refer to these schedules as the A-schedule and the B-schedule respectively. Note that the numbers n_A and n_B are allowed to become negative, which accounts for the empty tasks that are inserted. o^A indicates the direction of the task that starts at the critical time c^A. The symbol $\bar{X}$ denotes the inverse of X, i.e. $\bar{A} = B$ and $\bar{B} = A$.

4.2.1 Forbidden Zones

Using the two critical times c^A and c^B two pairs of forbidden zones can be defined FA^A, FB^A and FA^B, FB^B.

Consider the forbidden zones FA^B, FB^B and the critical time c^B (the case for FA^A, FB^A and c^A is analogous). Assume that $o^B = A$ (the direction of the first task in the backschedule ending on a B-task). Depending on the critical time c^B one of the following happens:

- $c^B < r$: It is not possible to schedule the tasks between r and d. Set $FA^B = (-\infty, \infty)$ and $FB^B = (-\infty, \infty)$.
- $c^B \in [r, r + 2)$: Define two new forbidden zones $FA^B = (c^B - 2, r)$ and $FB^B = (c^B - 1, r + 1)$.
- $c^B \geq r + 2$: No forbidden zone

Figure 2 shows two examples of the second case above.

The forbidden zones (FA^A, FA^B, FB^A, FB^B) cannot be used as the result of the backscheduling algorithm. It is the intersection of the forbidden zones that will define the eventual forbidden zones $FA = FA^A \cap FA^B$ and $FB = FB^A \cap FB^B$. If one of the backschedules fails (assuming the schedule ends on an X-task) then $FA^X = FB^X = (-\infty, \infty)$ and the intersection will be $FA = FA^{\bar{X}}$ and $FB = FB^{\bar{X}}$. If both backschedules fail, then the algorithm fails.

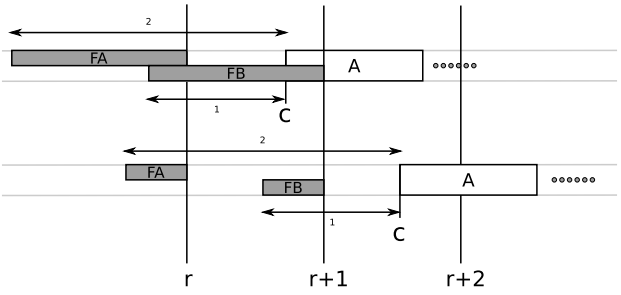

Fig. 2 Two examples of how forbidden zones are derived when $o^B = A$.

Lemma 1 *If $\#A \neq \#B$ then one of the two backschedules (for a given r and d) will count one more operation (tasks and empty lockings) than the other. Define c_{long} to be the critical time of the longer schedule (with more operations) and c_{short} to be the critical times of the shorter schedule. In this case $c_{long} \leq c_{short}$.*

Proof (Note that the last task will be in a different direction in the two schedules, so this lemma is not trivial.)

First we show that the first task in both schedules will be in the same direction. Assume $\#A > \#B$. For the A-schedule the last task is an A-task. The first task cannot be a B-task when scheduling with a minimal number of empty tasks (Algorithm 1): this would imply that there are an equal number of (real/empty) tasks in the backschedule, so we can leave off the first B-task since $\#B < \#A$. For the B-schedule the last task is a B-task. In this case the first task cannot be a B-task either: this would mean that there are more (real/empty) B-operations than A-operations, which is impossible since $\#B < \#A$.

So, for both schedules the first tasks have the same direction. This automatically implies that there will be one schedule that is one operation longer than the other, so we can define the critical times as c_{long} and c_{short} stated in the lemma.

The proof continues by induction. Assume that the long schedule ends on a B-task and the short schedule ends on an A-task (the other case is analogous). In the first step of the induction ($i = 0$) we take an empty schedule as short schedule (so $c^0_{\text{short}} = d$) and in the long schedule a B-task is put at the end (so $c^0_{\text{long}} \leq d - 1$). It is clear that $c^0_{\text{long}} < c^0_{\text{short}}$.

Assume $c^{i-1}_{\text{long}} < c^{i-1}_{\text{short}}$. Now add at the beginning of both the short and long schedule an extra task T_i (in

the same direction). In this case $c^i_{\text{short}} \leq c^{i-1}_{\text{short}} - 1$ and $c^i_{\text{long}} \leq c^{i-1}_{\text{long}} - 1$.

T_i can certainly start at time c^i_{long} in the short schedule: at that time a task in the same direction starts in the long schedule, which excludes the existence of forbidden zones. Also $c^i_{\text{long}} \leq c^{i-1}_{\text{long}} - 1 < c^{i-1}_{\text{short}} - 1$ which excludes overlap with previously scheduled tasks in the short schedule. This proves that $\min(s^i_{\text{short}}) = c^i_{\text{long}}$ and that $c^i_{\text{long}} \leq c^i_{\text{short}}$, which proves the induction step. □

Example 2 Assume $\#A \neq \#B$. Figure 3 shows a situation in which $\#A = 2$ and $\#B = 0$. The above schedule, which only counts 3 operations is clearly shorter than the one below in which an extra empty B-locking is inserted to ensure the schedule ends on a B-schedule. The light grey operations indicate the sequence that both the short and long schedules have in common. It's impossible that the first schedule takes longer than the second, since if all operations start at the same time as the corresponding operation in the second schedule, then $c_{\text{short}} = c_{\text{long}}$.

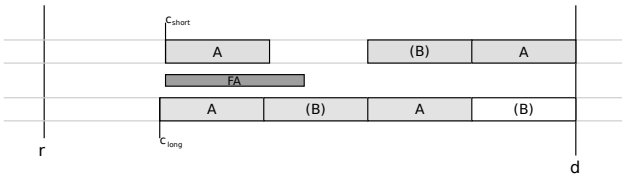

Fig. 3 A short and long schedule.

Using Lemma 1 the search for the intersection of the forbidden zones can be simplified (still assuming $\#A \neq \#B$). The forbidden zones belonging to the short schedule will be a subset of the forbidden zones belonging to the long schedule. As such, the forbidden zones belonging to the long schedule can be discarded. So

$$FA^{\text{short}} = (c_{\text{short}} - 2, r) \subseteq FA^{\text{long}} = (c_{\text{long}} - 2, r) \quad (1)$$

and

$$FB^{\text{short}} = (c_{\text{short}} - 1, r + 1) \subseteq FB^{\text{long}} = (c_{\text{long}} - 1, r + 1) \quad (2)$$

and $FA = FA^{\text{short}} \cap FA^{\text{long}} = FA^{\text{short}}$ and similar for FB.

4.2.2 More Forbidden Zones

The forbidden zones mentioned above are not the only ones that can be defined. Consider again the case $\#A \neq \#B$, on the interval $[r, d]$. Assume the first task T_1 has $o_1 = A$.

As stated before forbidden zones $FA = (c-2, r)$ and $FB = (c - 1, r + 1)$ can be defined. There is however a sequence of zones. Consider the set of zones

$$\{FA_k\}_k = \{(c-2+2k, r+2k) : k \in \mathbb{N}, r+2k < d\}_k \quad (3)$$

and the set

$$\{FB_k\}_k = \{(c-1+2k, r+1+2k) : k \in \mathbb{N}, r+1+2k < d\}_k \quad (4)$$

All of these zones are valid forbidden zones, since scheduling an A-task in (for example) $(c, r+2)$ implies that the other $\#A-1$ A-tasks cannot be scheduled between $[r, d]$ anymore.

These extra forbidden zones are however not necessary for the correct execution of the algorithm, as will be shown later. It does remain an interesting topic for future research to consider if it is possible to construct an alternative algorithm that takes these extra zones into account, because this might provide a considerable speedup. In the remainder of this paper we will ignore these additional forbidden zones.

4.2.3 The case $\#A = \#B$

The case $\#A = \#B$ can be solved by running backscheduling twice on the same interval $[r, d]$ with respectively one A task less or one B task less. From the proofs later on, it is clear that the correct execution of the algorithm does not depend on backscheduling for $\#A = \#B$.

4.2.4 Equivalent Release Times

Assume $\#A \neq \#B$ and that both the short and long backschedule start with an A-task. If $\exists FA \in \mathcal{F}_A : r \in FA$, then it follows that the A-task at the start of the backschedule cannot start in $[r, \sup(FA))$. Define the equivalent release time $r^A_{\text{eq}} = \sup(FA)$ [1]. In this case backscheduling can be executed on the interval $[r^A_{\text{eq}}, d]$ instead of $[r, d]$. Equivalent release times will play a crucial role in the correctness proof and the backscheduling algorithm.

Example 3 Figure 4 shows a situation in which $r \in FA$ when backscheduling on $[r, d]$. Since $c \in (r, r + 2)$ no forbidden zones should have to be added according to the classical definition of forbidden zones. However, $c \in (r_{\text{eq}}, r_{\text{eq}} + 2)$ so forbidden zones have to be added by using an equivalent release time. These new forbidden zones (computed with r_{eq} instead of r) are indicated in dashed lines.

[1] The superscript A refers to the direction of the first task in the backschedule

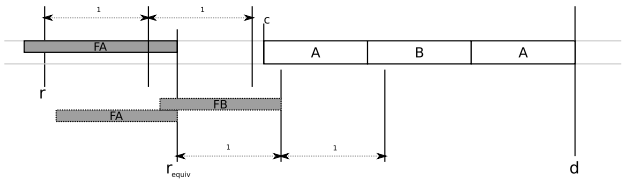

Fig. 4 Backscheduling with equivalent release times when $r \in FA$. At the top the classical way of scheduling is shown, resulting in no additional forbidden zones. At the bottom the backschedule using an equivalent release time is shown, which results in two forbidden zones.

4.2.5 Complete Set of Forbidden Zones

Throughout the remainder of the text we will refer to finding 'all' forbidden zones in the backscheduling algorithm. At this point we however do not provide a complete algorithm yet. Instead we provide a simple definition that verifies if a given set of forbidden zones $\mathcal{F}_A$, $\mathcal{F}_B$ is complete. This definition is used in the correctness proof of the forward scheduling algorithm, to prove that if backscheduling returns a set of forbidden zones that matches this definition, a valid schedule will be constructed by forward scheduling. The definition can also be used to verify the correctness of the complete backscheduling algorithm given later on.

Definition 4 (Complete set of forbidden zones) For an arbitrary subset of tasks $\mathcal{S}$ and a set of forbidden zones $\mathcal{F}_A$, $\mathcal{F}_B$, define:

- $r_{\min} \leftarrow \min(r_i : T_i \in \mathcal{S})$
- $d = \max(d_i : T_i \in \mathcal{S})$
- $n_A = \#\{T_i \in \mathcal{S} : o_i = A\}$ and $n_B = \#\{T_i \in \mathcal{S} : o_i = B\}$
- $r^A_{\text{eq}} = \begin{cases} \sup(F) & \text{if } \exists F \in \mathcal{F}_A : r_{\min} \in F \\ r_{\min} & \text{else} \end{cases}$
- $r^B_{\text{eq}} = \begin{cases} \sup(F) & \text{if } \exists F \in \mathcal{F}_B : r_{\min} \in F \\ r_{\min} & \text{else} \end{cases}$
- $c^A, o^A, c^B, o^B =$ B.S. subprocedure on $(r^A_{\text{eq}}, d, \mathcal{S}, \mathcal{F}_A, \mathcal{F}_B)$ when $n_A > n_B$
- $c^A, o^A, c^B, o^B =$ B.S. subprocedure on $(r^B_{\text{eq}}, d, \mathcal{S}, \mathcal{F}_A, \mathcal{F}_B)$ when $n_A < n_B$

A set of forbidden zones $\mathcal{F}_A$, $\mathcal{F}_B$ (e.g. from the backscheduling algorithm) is said to be complete if for every subset $\mathcal{S}$ of the tasks set $\mathcal{T}$, the following assertions are satisfied:

$$\text{if } n_A > n_B \begin{cases} c^A \geq r^A_{\text{eq}} \\ (c^A - 2, r^A_{\text{eq}}) \subset \bigcup \mathcal{F}_A \\ (c^A - 1, r^A_{\text{eq}} + 1) \subset \bigcup \mathcal{F}_B \end{cases}$$

$$\text{if } n_A < n_B \begin{cases} c^B \geq r^B_{\text{eq}} \\ (c^B - 2, r^B_{\text{eq}}) \subset \bigcup \mathcal{F}_B \\ (c^B - 1, r^B_{\text{eq}} + 1) \subset \bigcup \mathcal{F}_A \end{cases}$$

4.3 Forward Scheduling

Once all forbidden zones have been found in the backscheduling phase, the forward scheduling can start. From now on, we will assume that all the forbidden zones [2], as defined in the previous section, have been found. An algorithm to do this is given later in Section 4.5.

Algorithm 2 shows the forward scheduling algorithm. In the forward scheduling phase, A and B-tasks must be scheduled alternating. Define $X \in \{A, B\}$ as the direction of the next task to be planned and t the end time of the last scheduled task. If there is a forbidden zone around time t, however, we should shift t, which is done in line 3. The next task is the task T_j in direction X with the earliest deadline and $r_j \leq t$, which is determined in line 6. If such a task exists, it can be scheduled, if not, the algorithm chooses between scheduling an empty task at time t or waiting for the next available task in direction X (with $r_j > t$). This is done in lines 10 to 21: $r_{\text{equiv}}+1$ is the time that a consecutive task (direction $\bar{X}$) can start if the task T^X_j is scheduled. $r_{\bar{X}}$ is the time that a consecutive task can start if an empty locking is scheduled. The option that allows the consecutive task to start the earliest is chosen.

In this way a schedule is obtained that maps all tasks T_j to a starting time s_j and that inserts empty tasks $L_l = (s_l, x_l)$ that start at time s_l in direction $x_l \in \{A, B\}$.

4.4 Correctness Proof

A formal correctness proof is given, based on the assumption that all forbidden zones (as defined above) have been found. It is shown that, if all zones are found in the backscheduling phase and no failure occurs in that phase, the forward scheduling algorithm will create a valid schedule. The general construction of the proof is a proof by contradiction: if we come across a failure in the forward scheduling algorithm (which means that a task has to be scheduled past its deadline) then it can be shown that the backscheduling algorithm did not function correctly and did not add a forbidden zone.

First, some definitions are needed to classify all the intervals in a (partially) constructed schedule:

Definition 5 An interval $\tilde{F} = (a, b)$ is an effective forbidden zone in direction X in the schedule S if:

- $\exists\, F_i \in \mathcal{F}_X : a \in F_i \wedge b = \sup(F_i)$, and
- $\exists\, T_j : o_j = X \wedge b = s_j$ or $\exists L_l : o(L_l) = X \wedge b = s_l$, and

[2] This does not include the extra forbidden zones from Section 4.2.2.

Algorithm 2 Forward scheduling algorithm

1: $t \leftarrow 0$. Find the task T_1^A in direction A with smallest r and the task T_1^B in direction B with the smallest r. Check, using forbidden zones, which task can start the earliest. Set X to the direction of this task and t to its release time.
2: **while** not all tasks are scheduled **do**
3: **if** $\exists i : t \in FX_i$ **then**
4: $t \leftarrow sup(FX_i)$.
5: **end if**
6: Find the task T_j with $o_j = X$ that has not been scheduled yet with $r_j \leq t$ and d_j minimal.
7: **if** such a task T_j exists **then**
8: T_j: $s_j = t$, $t \leftarrow t + 1$ and $X \leftarrow \bar{X}$.
9: **else**
10: find the task T_j with $o_j = X$ that has not been scheduled yet with r_j minimal
11: **if** such a task T_j does not exist **then**
12: Schedule an empty task: $t \leftarrow t + 1$, $X \leftarrow \bar{X}$
13: Continue
14: **end if**
15: Compute r_{equiv}:

$$r_{\text{equiv}} \leftarrow \begin{cases} \sup(FX_i) & \text{if } \exists FX_i : r_j \in FX_i \\ r_j & \text{else} \end{cases} \quad (5)$$

16: Compute $r_{\bar{X}}$:

$$r_{\bar{X}} \leftarrow \begin{cases} \sup(F\bar{X}_i) & \text{if } \exists F\bar{X}_i : t+1 \in F\bar{X}_i \\ t+1 & \text{else} \end{cases} \quad (6)$$

17: **if** $r_{\text{equiv}} + 1 > r_{\bar{X}}$ **then**
18: Schedule an empty task: $t \leftarrow t + 1$, $X \leftarrow \bar{X}$
19: **else**
20: Schedule the task T_j: $s_j \leftarrow r_{\text{equiv}}$. $t \leftarrow r_{\text{equiv}} + 1$, $X \leftarrow \bar{X}$.
21: **end if**
22: **end if**
23: **end while**

– $\forall\, T_j : (s_j, s_j + 1) \cap (a, b) = \varnothing$.

An effective forbidden zone is a part of a forbidden zone in which no task is executed and that is followed immediately by a task in the same direction as the forbidden zone. In the following we use $\tilde{\mathcal{F}}_X$ to denote the set of effective forbidden zones in direction X (omitting the reference to the specific schedule S). We define $\tilde{\mathcal{F}} = \tilde{\mathcal{F}}_A \cup \tilde{\mathcal{F}}_B$.

Definition 6 An interval $I = (a, b)$ is an *idle*-zone if:

– $\forall\, T_j : (s_j, s_j + 1) \cap (a, b) = \varnothing$, and
– $\forall\, \tilde{F}_i \in \tilde{\mathcal{F}} : \tilde{F}_i \cap (a, b) = \varnothing$

An *idle*-zone is an interval in which no (part of a) task is executed and that is not (part of) an effective forbidden zone.

Example 4 Before presenting a formal proof, we first demonstrate the general structure of the proof with a simple example. Figure 5 (top) shows a schedule generated by forward scheduling. The last task T_j with $o_j = A$ is late in this schedule, since the deadline d_j is exceeded. All other tasks are on time. The minimal starting time of all the A-tasks is $r_{\min}$. There is also one forbidden zone FB of which a part is an effective forbidden zone.

Figure 5 (bottom) shows what would happen if backscheduling was applied on the late task T_j and all other previous A-tasks which are on time. It is clear that backscheduling fails: the first A-task would have to start before $r_{\min}$. This is a contradiction, since the backscheduling algorithm has to succeed before starting forward scheduling.

In this example there are no empty A-lockings, idle zones... and we silently assumed that all A-tasks have a deadline smaller than d_j. In the following formal proofs these cases are considered.

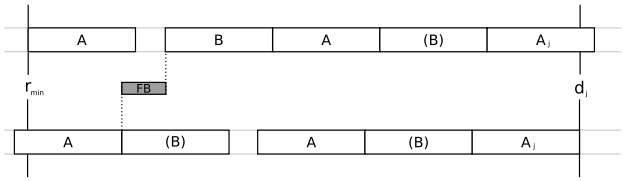

Fig. 5 Illustration of the proof of Lemma 2.

In Lemma 2 a simple proof is given, for the cases that no empty tasks and idle zones are present in the partial schedule. Later on, this is extended to include empty tasks and idle zones.

Lemma 2 *Consider a (partial) schedule S constructed by the forward scheduling algorithm applied to a set of complete forbidden zones. Assume that in this schedule there is a task T_j that is late ($s_j + 1 > d_j$) and assume that $\forall\, T_i, s_i < s_j, o_i = o_j : d_i \leq d_j$. Define $r_{min} = \min_{o_i = o_j}(r_i)$. Assume also that there are no empty lockings in direction o_j and that there are no idle-zones in the interval (r_{min}, d_j). In this case the back-scheduling algorithm should have failed on the interval (r_{min}, d_j) (i.e. the output does not satisfy Def. 4).*

Proof Assume $o_j = A$ (the proof for $o_j = B$ is analogous).

All the A-tasks that start in the interval $(r_{\min}, d_j)$ are, given the assumptions made, real (not empty) tasks. We now apply backscheduling on the interval $(r_{\min}, d_j)$ with all these A-tasks. This is allowed because $\forall T_i, s_i < d_j, o_i = A : r_{\min} \leq r_i < d_i \leq d_j$. Renumber the A-tasks T_i by increasing starting time s_i as $T'_1, T'_2, ..., T'_k$ ($T'_k = T_j$).

Backscheduling will assign all these tasks a new starting time s'_i that is strictly smaller than the starting

time in the original schedule (s_i is the starting time of T'_i in the original schedule). This is clearly the case for $T'_k = T_j$, since $s'_k < d_j - 1$ and T_j was late in the original schedule ($s_k > d_j - 1$). By induction it is easy to show that all other tasks will be assigned a lower starting time than in the original schedule: assume $s'_l < s_l$ (induction assumption). We will show that the property also holds for $s'_{l-1} < s_{l-1}$.

- First a B-task (empty) has to be inserted before the task T'_l (to ensure the alternation of directions). It is clear that the starting time of this B-task is lower than the starting time of the matching B-task in the original schedule: the new starting time will be $s'_l - 1$ or, if $\exists FB_m : s'_l - 1 \in FB_m$ it will be $\inf(FB_m)$.
- Next, the task T'_{l-1} is inserted. By the same reasoning as the B-task, it will have a lower starting time than in the original schedule.

Note that there are no idle zones and no empty A-tasks. Empty B-tasks do not cause trouble, as only the A-tasks are considered when backscheduling.

So, by induction, a new critical time $c' = s'_1 < s_1$ is obtained. One of the three following events happens at time $r_{\min}$ in the original schedule:

- If $\exists \tilde{FA} \in \tilde{\mathcal{F}}_A : r_{\min} \in \tilde{FA}$ then $c' \leq \inf(\tilde{FA}) < r_{\min} < sup(\tilde{FA}) = s_1$. This is clearly a contradiction, as $c' < r_{\min}$. This indicates the backscheduling algorithm should have failed.
- If $\exists T_b, o_b = B : r_{\min} \in [s_b, s_b + 1)$ then for the new schedule $c = s'_1 < s_b + 1$. The case that $c < r_{\min}$ is clearly a contradiction. In the other case $c \in [r_{\min}, s_b + 1)$, which implies the existence of a forbidden zone $FB = (c - 1, r_{\min} + 1)$. Another contradiction arises, since $s_b \in FB$.
- In all other cases the first A-task will start immediately at time $r_{\min}$ in the original schedule. Since $c' < s_1 = r_{\min}$, the backscheduling phase should have failed.

□

In Lemma 2 contradictions are shown by shifting tasks forward, thereby constructing a backschedule that shows the existence of forbidden zones that have not been detected correctly. In Lemma 3 this is extended to situations in which the assumption $\forall T_i, s_i < s_j, o_i = o_j : d_i \leq d_j$ is dropped.

Lemma 3 *Consider a (partial) schedule S constructed by the forward scheduling algorithm applied to a set of complete forbidden zones. Assume that in this schedule there is a task T_j that is late ($s_j + 1 > d_j$) and assume that there is at least one task T_l with $s_l < s_j$, $o_l = o_j$ and $d_l > d_j$. Assume also that there are no empty lockings in direction o_j and that there are no idle-zones in the interval (s_l, d_j). In this case the backscheduling algorithm should have failed.*

Proof Assume $o_j = A$ (the proof for $o_j = B$ is analogous). If there are multiple tasks that match the conditions for T_l, the task with the highest starting time s_l is taken.

This task was chosen at the time s_l from all A-tasks that were available at that time, because T_l had the lowest deadline. Since $d_l > d_j$ this implies that $\forall T_k, o_k = A, s_l < s_k \leq s_j : r_k > s_l$ (if there was a task T_k with $r_k < s_l$, that task should have been started at s_l instead of T_l). Let r be the smallest release time of the tasks T_k (with $o_k = A, s_l < s_k \leq s_j$). In this case backscheduling can be applied on the interval (r, d_j), just like in Lemma 2, to show a contradiction. □

What remains to be done is adding idle zones and empty lockings in the direction of the 'late' task.

Lemma 4 (Empty lockings lemma) *Consider, again, a (partial) schedule S constructed by the forward scheduling algorithm applied to a set of complete forbidden zones. Assume that in this schedule there is a task T_j that is late ($s_j + 1 > d_j$) and that, without loss of generality, $o_j = A$. Assume that there is an empty locking L with $o(L) = A$ and that there are no idle zones or empty lockings (with direction A) in (s_L, d_j). Assume that $\forall T_i, o_i = A, s_L < r_i < d_j : d_i \leq d_j$. In this case the backscheduling algorithm should have failed.*

Proof From the construction of the forward scheduling algorithm it follows that, for an empty locking to occur, there should be no tasks T_i available in direction A at time s_L. Or, formally:

$$\forall T_i, o_i = A, s_L < s_i < s_j : r_i > s_L \tag{7}$$

Now consider line 17 of Algorithm 2 at time $t = s_L$, which states that $r_{\text{equiv}} + 1 > r_{\bar{X}}$ and line 16 which implies that $r_{\bar{X}} \geq s_L + 1$.

Two different situations can arise:

1. The case that L is followed immediately by a B-task (see Fig. 6). Apply backscheduling to all tasks T_i in direction A with $s_L < s_i < s_j$ on the interval $(r_{\min}, d_j)$. This is allowed since $s_L < r_{\min} \leq r_i < d_j$ and $s_L + 1 < d_i < d_j$. The first A task that follows L is assigned a new starting time $s' < s_L + 2$ in this backschedule [3]. This implies the existence of a forbidden zone $FA_{\text{new}} = (s' - 2, r_{\min})$. This implies $s_L \in FA_{\text{new}}$, which is impossible as no A-task can start in such a forbidden zone.

[3] Note that if there is an effective forbidden zone $\tilde{FA}$ between the first B task and the A task, then the A-task would shift to $\inf(\tilde{FA}) < s_L + 2$

2. The case that L is followed by an effective forbidden zone $\tilde{FB}_p \subset FB_q$. Since $s_L + 1 \in FB_q$, it follows that $r_{\bar{X}} = \sup(\tilde{FB}_p) = \sup(FB_q)$ and that $r_{\text{equiv}} > \sup(\tilde{FB}_p) - 1$.
   Backscheduling is applied to all A-tasks on the interval (r_{equiv}, d_j). Since there are no idle-zones the A-task will shift forward and will get a new starting time $s' < \sup(\tilde{FB}_p) + 1$. This all implies that

$$r_{\text{equiv}} + 2 > \sup(\tilde{FB}_p) + 1 > s' \tag{8}$$

   which will give rise to a new forbidden zone $FB_{\text{new}} = (s' - 1, r_{\text{equiv}} + 1)$. Since $\sup(\tilde{FB}_p) \in FB_{\text{new}}$ this is a contradiction: the B-task starts at time $\sup(\tilde{FB}_p)$, which is not allowed by the forbidden zones.

□

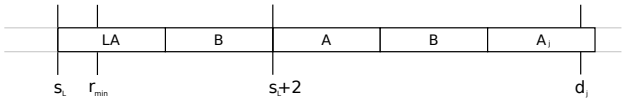

Fig. 6 Illustration of the proof of Lemma 4 (case 1).

Lemma 5 (Idle zone lemma A) *Consider, again, a (partial) schedule S constructed by the forward scheduling algorithm applied to a set of complete forbidden zones. Assume that in this schedule there is a task T_j that is late ($s_j + 1 > d_j$) and that, without loss of generality, $o_j = A$. Assume that there is an idle zone I that starts at time $\inf(I)$ and is followed by an effective forbidden zone FA_I (which might have zero length, e.g. there might not be a forbidden zone), which is followed by a task T_i ($o_i = A$). Assume that $\forall T_i, o_i = A, \inf(I) < r_i < d_j : d_i \leq d_j$. Assume there are no other idle zones or empty lockings (with direction A) in $(\inf(I), d_j)$. In this case the backscheduling algorithm should have failed.*

Proof (See Fig. 7). Since there is an idle zone, the definition of the forward scheduling algorithm guarantees that there were no available A-tasks at time $\inf(I)$, so:

$$\forall T_k, o_k = A, \inf(I) < s_k \leq s_j : \; r_k \geq \inf(I). \tag{9}$$

Define $r_q = \min\{r_k : \inf(I) < r_k, o_k = A\}$. When backscheduling on (r_q, d_j) a failure will occur, since $s' < r_q$ with s' the new starting time of the first A task. □

Lemma 6 (Idle zone lemma B) *Consider, again, a (partial) schedule S constructed by the forward scheduling algorithm applied to a set of complete forbidden zones. Assume that in this schedule there is a task T_j that is late ($s_j + 1 > d_j$) and, without loss of generality, that $o_j = A$. Assume that there is an idle zone I that starts at time $\inf(I)$ and is followed by a forbidden zone FB_I (which might have zero length, e.g. there might not be a forbidden zone), which is followed by a task T_i ($o_i = B$). Assume there are no other idle zones or empty lockings (with direction A) in $(\inf(I), d_j)$. Assume that $\forall T_k, o_k = A, \inf(I) < r_k < d_j : d_k \leq d_j$. In this case either the forward or backward scheduling algorithm was not executed correctly or, when applying backscheduling with deadline d_j the first A-task that follows the idle zone is shifted before $\inf(I)$.*

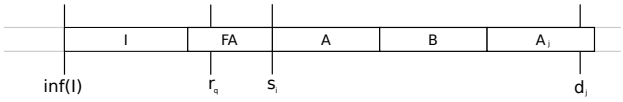

Fig. 7 Illustration of the proof of Lemma 5.

Proof Because of the presence of the idle zone, it is known from the definition of the forward scheduling algorithm (at time $t = \inf(I)$ and with $X = B$) that $r_{\text{equiv}} = \sup(FB_I) = s_i$ and that $r_{\text{equiv}} + 1 < r_{\bar{X}}$. Note that r_{equiv} is associated with the B-task T_i and not an A-task like in the previous lemma.

There are two possible situations that can arise:

1. $\exists FA_p : \inf(I) + 1 \in FA_p$: in this case $r_i + 1 < \sup(FA_p)$ and $r_{\bar{X}} = \sup(FA_p)$. It follows that

$$(\inf(I) + 1, r_i + 1) \subset FA_p \tag{10}$$

   When backscheduling all A-tasks on an interval ending with the deadline d_j, the first A-task after the idle-zone will be assigned a new starting time $s'_A < \inf(I) + 1$ because of the forbidden zone $(\inf(I) + 1, r_{\text{equiv}} + 1)$.
   Note that the idle zone I is either preceded directly by another A-task or by an effective forbidden zone. In any case the A-task that precedes I will be shifted forward when backscheduling: there must be a B task between the two A-tasks, which will be pushed forward by the lowered starting time s'.
2. $\nexists FA_p : \inf(I) + 1 \in FA_p$: in this case $r_{\text{equiv}} + 1 < \inf(I) + 1$, which is impossible since in that case $r_B \leq r_{\text{equiv}} < \inf(I)$. This contradicts the fact that an idle zone can only occur if there are no B-tasks available (i.e. $r_B > \inf(I)$). □

Theorem 2 *Assume that the backscheduling algorithm returns a complete set of forbidden zones (according to Def. 4), then the forward scheduling algorithm will always succeed and find a valid schedule.*

Proof Assume that there is a late task T_j in a schedule (constructed by the execution of backscheduling and forward scheduling) that starts at time s_j and has deadline d_j ($s_j > d_j - 1$). Assume that this task has $o_j = A$ (the proof for B is analogous).

By starting at the deadline d_j and moving left (to lower times) one of the following can happen:

- If there is a task T_i in the backschedule with $d_i > d_j$ a contradiction occurs by Lemma 3.
- If an idle zone followed by an A-task is encountered this yields a contradiction by Lemma 5.
- If an empty locking in direction A is encountered this yields a contradiction by Lemma 4.
- If none of the above are encountered a contradiction is yielded by application of Lemma 2 (combined with Lemma 6 if an idle zone followed by a B-task is encountered).

So, if a task is late, this implies that the backscheduling phase was not executed correctly (i.e. it should have failed). This is a contradiction with the assumption made. □

4.4.1 Remark on the Case $\#A = \#B$

Note that in the proofs backscheduling is always applied in situations where $\#A \neq \#B$. This again confirms that it is not necessary to apply backscheduling in the cases that $\#A = \#B$.

4.5 Backscheduling

Theorem 2 proves that the forward scheduling algorithm will succeed provided that the backscheduling algorithm finds all forbidden zones (as defined in the previous sections) and does not encounter a failure (when $c < r$).

A naive way to find all these zones would be to sort all release times so that $r_1 \leq r_2 \leq \ldots r_n$ and then apply backscheduling (Algorithm 1) on every interval (r_i, d_j) with the r_i's going from high to low. Besides being very inefficient (backscheduling is started from scratch every time) it is also incorrect: all forbidden zones need to be discovered. Since every execution of the backscheduling algorithm can also introduce forbidden zones in $(c_i - 1, r_i + 1)$, this forbidden zone might influence other (previous, meaning for higher r_i) executions of the backscheduling algorithm. A task that started at a certain time in the previous backschedule might now start inside the newly defined forbidden zone, which is not allowed. A second problem is the creation of equivalent release times: a release time might fall inside one of the new forbidden zones, causing it to be replaced by an equivalent time.

First it will be shown that the first problem stated above (tasks falling inside a new forbidden zone) will not occur. The newly defined forbidden zone cannot influence previous executions of the backscheduling algorithm. In the original problem described by Garey et al (1981) this 'influencing' problem does not occur, since forbidden zones were only defined outside the current backscheduling interval. So, the newly defined forbidden zones could definitely not influence previous executions of backscheduling (higher r_i). [4] The second problem stated (creation of equivalent release times) can, however, occur.

In this section we use a new notation for the release times: with r_i^A we denote the release time r_i or, if $r_i \in FA_k$, we assume that it is automatically replaced by $\sup(FA_k)$, which will be the equivalent release time. The definition of r_i^B is analogous.

In Section 4.5.1 the non-influencing theorem is proven, solving the first problem stated above. Next, the backscheduling algorithm is stated. In the derivation of the complexity of the algorithm the second problem (creation of equivalent times) is discussed. An upper bound on the number of times a release time can be replaced ensures that the algorithm will run in polynomial time in the number of ships.

4.5.1 Non-influencing Theorem

To prove the statements made in the introduction, we consider two release times $r_1 = r_i^X$ and $r_2 = r_j^Y$, assuming $r_2 < r_1$. All forbidden zones for r_1 are known, and the proof considers what happens when the zones belonging to r_2 are added. To prove this theorem, it is sufficient to show that it is impossible that $c_1 \in FX_2$.

Theorem 3 *Consider the forbidden zones FA_1 and FB_1 from the execution of backscheduling on the interval (r_1, d_1) (with $r_1 = r_i^X$ s.t. $\nexists FX_k : r_i^X \in FX_k$) and a new execution on the interval (r_2, d_2) (with $r_2 = r_j^Y$ s.t. $\nexists FY_k : r_j^Y \in FY_k$). In this case the forbidden zones belonging to (r_2, d_2) will not cause any change in the backscheduling on (r_1, d_1), i.e. it will not change the critical time c_1 (but might create a new equivalent release time replacing r_1).*

[4] In 4.2.2 'extra' forbidden zones are defined that could be used to construct an alternative algorithm. One would suspect that using these extra zones would cause a massive number of these 'influencing' problems, in which the newly defined forbidden zones influence previous executions of backscheduling. The results stated here however, hint otherwise. It is an open problem to check if our results can be extended to these 'extra' forbidden zones and by doing this showing even more of the underlying structure of the problem.

Proof Assume without loss of generality that the first task in the (r_2, d_2) backschedule is an A-task.

The forbidden zone $FA_2 = (c_2 - 2, r_2)$ that is added (if $c_2 < r_2 + 2$)) cannot influence the backscheduling in (r_1, d_1) in any way. So, only the other forbidden zone $FB_2 = (c_2 - 1, r_2 + 1)$ needs to be investigated further.

Consider two cases:

- The first task belonging to the (r_1, d_1)-backschedule is an A-task. The forbidden zone FB_2 cannot influence the backschedule on (r_1, d_1) in any way. The first B-task in that schedule cannot start until after $r_1 + 1 > r_2 + 1 > FB_2$.
- The first task belonging to the (r_1, d_1)-backschedule is a B-task. Assume FB_2 influences backscheduling on (r_1, d_1). It must be that $c_1 \in FB_2$, so:

  $$c_2 - 1 < c_1 < r_2 + 1 < r_1 + 1 \tag{11}$$

  This implies the existence of a forbidden zone $FA_1 = (c_1 - 1, r_1 + 1)$, as $c_1 < r_1 + 1$. Because $c_1 - 1 < r_2$, $r_2 \in FA_1$. So, r_2 should have been replaced by a higher equivalent release time $r_2' = \sup(FA_1)$. This is a contradiction, so the assumption that $c_1 \in FB_2$ is wrong.

From both cases it clearly follows that no change will happen in the forbidden zones belonging to (r_1, d_1). □

Based on this theorem it suffices to guarantee that if $r_1 < r_2$, the backscheduling for r_2 is always executed before the backscheduling on r_1. If one of the release times r_i is replaced by an equivalent release time, we should replace the original release time by the new one and reorder the list of r_i's if necessary. Note that, notationwise, the index is preserved when reordering.

4.5.2 Backscheduling Algorithm

We now state the final backscheduling algorithm in Algorithm 3. Backscheduling is applied for all r_i, in decreasing order. Because of equivalent release times, we have to create a lexicographically sorted set $R = \{(r^X_{i,\text{equiv}}, i, X)\}$ with $X \in \{A, B\}$ that stores the equivalent release times and the direction for which they apply. This list is initialized with the r_i (stored twice, once for every direction) in line 5.

For each r_i (line 7) backscheduling is applied for each $d_j \geq d_i$. Instead of executing a full backscheduling every time, the result that belongs to (r_i, d_j) can be reused to compute the backschedule for (r_{i-1}, d_j): an extra task is added. To be able to reuse this result we define the tuples

$$I_{i,j} = (c^A_{i,j}, c^B_{i,j}, \#A_{i,j}, \#B_{i,j}) \tag{12}$$

in which $c^A_{i,j}$ and $c^B_{i,j}$ are the critical times of the backschedule on (r_i, d_j) ending on an A task and a B task respectively. $\#A_{i,j}$ and $\#B_{i,j}$ are the number of effective lockings in the respective direction for the backschedule on (r_i, d_j). The critical time subroutine (line 8 and Appendix A) iterates over all $d_j > d_i$ for a certain r_i, updates the $I_{i,j}$ and returns a critical time. Lines 9-14 create the new forbidden zones. Lines 15-23 look for the creation of new equivalent release times.

Algorithm 3 Backscheduling algorithm

```
1: Restart ← true
2: F_A ← F_B ← {}
3: while Restart = true do
4:    Restart ← false
5:    Initialize the sorted set R ← {(r_i, i, A), (r_i, i, B)}_i
6:    I_{i,j} ← undefined
7:    for (r^X_{i,equiv}, i, X) from R in decreasing order do
8:       C ← Critical time subroutine(r^X_{i,equiv}, i, X)
9:       if C < r^X_{i,equiv} then
10:         stop (FAIL)
11:      else if C < r^X_{i,equiv} + 2 then
12:         Add a forbidden zone (C − 2, r^X_{i,equiv}) to F_X
13:         Add a forbidden zone (C − 1, r^X_{i,equiv} + 1) to F_X̄
14:      end if
15:      for (r^Y_{l,equiv}, l, Y) from R do
16:         if ∃F ∈ F_Y : r^Y_{l,equiv} ∈ F then
17:            r^Y_{l,equiv} ← sup(F). (Reorder R if necessary)
18:            Restart ← true
19:         end if
20:      end for
21:      if Restart = true then
22:         break
23:      end if
24:   end for
25: end while
```

4.5.3 Correctness

Due to Thm. 3 it has become very easy to demonstrate the correctness of the backscheduling algorithm, i.e. that it returns a complete set of forbidden zones as in Def. 4 (or fails). Removing all the elements from the algorithm that are due to the incremental build-up of the backschedules (i.e. $I_{i,j}$), what remains is almost an exact copy of the definition of the forbidden zones. The non-influencing theorem (Thm. 3) guarantees that newly created forbidden zones don't influence previous backschedules. Also the creation of new equivalent release times is detected and results in a restart of the algorithm (preserving forbidden zones however), which ensures that all possible equivalent release times were taken into account when the algorithm terminates. In the next part we show that the number of creations of

new equivalent release times is finite and as such the algorithm will always terminate.

4.5.4 Complexity

Time complexity It can be easily checked that the critical time subroutine runs in $O(n \log n)$: the outer loop adds $O(n)$ to the complexity and looking for FA_k and FB_k is $O(\log n)$ provided that forbidden zones are stored sorted by their starting time and merged in case of overlap.

The complexity of the entire backscheduling algorithm is harder to determine. Excluding the restarting of the algorithm (the outer while-loop in algorithm 3 which can be triggered by line 22) the complexity is $O(n^2 \log n)$. The for-loop over R contributes $O(n)$, the call to the critical time subroutine $O(n \log n)$. The inner loop over R at line 15 also has a complexity of $O(n \log n)$. All other steps in the algorithm have a lower complexity, so they can be neglected.

To determine the complexity when including line 22, one needs an upper bound on the number of times a release time is replaced by an equivalent time.

Definition 7 If, during backscheduling, a new forbidden zone FX_k is added such that $r_i^X \in FX_k$ (which means that r_i^X should be replaced by the equivalent time $\sup(FX_k)$) we say that a 'replacement' event has occurred. We call this an E event.

Lemma 7 *The number of replacement events $\#E$ is bounded by $2\#R_c$, with R_c*

$$R_c = \left\{ r_i + k \,|\, r_i + k < \max_i(d_i),\, k \in \mathbb{N} \right\} \tag{13}$$

and

$$\#E \leq 2\#R_c \leq 2n(\max_i(d_i) - \min_i(r_i)) \tag{14}$$

Proof Note that on replacement of a release time r_i^X, it is always replaced by $\sup(FX_k)$. Since $\sup(FX_k) \in \{r_j^Y, r_j^Y + 1\}_j$, the replacement is actually $r_i'^X \leftarrow r_j^Y$ or $r_i'^X \leftarrow r_j^Y + 1$ for a certain j. The set R_c is 'closed' under the replacement event: if a certain release time $r_i^X \in R_c$ is replaced by $\sup(FX_k)$ the new value will also be in R_c (or, if it is not, the backscheduling phase will fail since no schedule exists). Since all initial release times r_i are in R_c all equivalent release times will also be in R_c which concludes our proof. □

From Lemma 7 it follows that the total complexity of the backscheduling algorithm is at most $O(D\, n^3 \log n)$ with $D = (\max_i(d_i) - \min_i(r_i)) \geq n$.

This bound is however rather high, especially in 'sparse' scheduling problems, with a relatively low number of tasks compared to the total scheduling time.

Definition 8 If, during backscheduling, it occurs that r_i^X is replaced by r_j^X $(i \neq j)$ we say that a release time 'collapse' has occurred. We denote this event as E_1.

If a replacement of a release time r_i^X occurs by a new time which is not equal to a certain r_j^X, we call the replacement event E_2.

It is clear that an event E will either be an E_1 event or an E_2 event. In some cases E_1 and E_2 can happen simultaneously though.

Lemma 8 *An event E_1 can only occur $2(n-1)$ times during backscheduling.*

Proof For a certain direction X, a collapse event implies that $r_i'^X = r_j^X$, so in the new list R there will be two equal equivalent release times and the number of unique release times is reduced by 1. This can only happen $n-1$ times. The factor 2 comes from the fact that there are equivalent release times for both directions A and B. □

Lemma 9 *An event E_2 can only occur once for a certain r_i^X between two E_1 events (on any r_j^Y).*

Proof Assume an E_2 event happens, replacing r_i^A by r'^A_i. This implies that a FA_1 was added such that $r_i^A \in FA_1$. Because this was an E_2 event, this means that $\sup(FA_1) = r_l^B + 1 = r'^A_i$, for a certain T_l. So, also a forbidden zone $FB_1 = (c_l - 2, r_l^B)$ was added.

Assume now that, after an arbitrary number of E_2 events (and no E_1 events) on any r_i^X a new E_2 event happens, this time on $r_i'^A$. Again, since this is an E_2 event, forbidden zones FA_2 (take $\sup(FA_2) = r_m^B$) and $FB_2 = (c_m - 2, r_m^B)$ are added.

Because $\sup(FA_1) = r_i'^A = r_l^B + 1 \in FA_2$ this implies that $r_l^B \in FB_2$, causing a simultaneous E_1 event for r_l^B [5]. □

From both lemmas it follows easily that there can only be $O(n^2)$ E_1 and E_2 events in total. This gives a complexity of $O(n^4 \log n)$.

Memory complexity The forbidden zones occupy $O(n)$ memory. The total memory consumption of the backscheduling algorithm in its current form is $O(n^2)$, since the $I_{i,j}$ entries need to be stored. But this can be reduced to $O(n)$ by only storing the $I_{i,j}$ values for the current i and $i+1$, which are needed in the critical time subroutine.

[5] In the case that r_l^B was replaced by $r_l'^B$ because of one or multiple E_2 events (without any simultaneous E_1 event) the lemma still holds, but r_l^B is just replaced by $r_l'^B$ in the proof.

Forward scheduling algorithm The forward scheduling algorithm runs in $O(n \log n)$. All tasks are sorted by r_i, so it only needs to search the task with the lowest deadline. There is an $O(n)$ factor because there are n tasks to schedule and an $O(\log n)$ factor because of searching through the forbidden zones.

4.5.5 Applications

The algorithm for solving SLSP1 can be applied in several ways. If $\forall\, T_i : w_i = w_{\max} \wedge l_i = l_{\max}$, we can use the result from Theorem 1 to solve SLOPminimax.

Another application is when all tasks T_i have been grouped beforehand in groups G_k. A group G_k is a set of tasks T_i with $\forall\, T_i, T_j \in G_k : o_i = o_j$ and $\forall G_k : Fit(\{(l_i, w_i)\}_{i:T_i \in G_k}) = true$. Define $r(G_k) = \max(r_i : T_i \in G_k)$ and $d(G_k) = \min(d_i : T_i \in G_k) + \Delta t + 1$. Again, by bisection on Δt, the minimax problem $\min\max_i(s_i - r_i)$ can be solved, with the constraint that $\forall\, G_k, \forall\, T_i \in G_k : s_i = s(G_k)$.

5 Future Work

5.1 Alternative Approaches

López-Ortiz and Quimper (2011) provide an algorithm for unit-length jobs with arbitrary release times and deadlines that runs in $O(n^2)$ (for $m = 1$, i.e. a single machine). There approach is based on reducing the scheduling problem to a shortest path graph problem. It would be interesting to see if similar modifications as this paper makes to the algorithm of Garey et al (1981) can be applied. The approach of using a graph problem is however fundamentally different than the approach with forbidden zones, so converting the modifications would be complicated.

5.2 Parallel Locks

Simons (1983) developed an algorithm (the barrier algorithm) that can schedule n tasks with equal processing times, arbitrary release times and deadlines on m identical machines in $O(n^3 \log\log n)$. Note that the time complexity of the algorithm is independent of the number of machines.

The idea behind this algorithm is quite similar to the forbidden zone algorithm. Instead of looking for forbidden zones beforehand, they are determined dynamically in the barrier algorithm during forward scheduling. Barriers are created, that indicate that a task cannot start before a certain time, which is the same idea as using forbidden zones.

Extending to multiple machines is possible in this case because, given only the start times for tasks on m machines (i.e. without knowing on which machine the task starts in the schedule), the given schedule can easily be divided among the m machines. The tasks can be divided cyclically among the machines: every machine is simply assigned the available task with the lowest starting time and we iterate through all machines until no tasks are left.

An interesting question for future research is modifying the barrier algorithm for the single ship, single lock scheduling problem and extending it to multiple lock chambers in parallel.

5.3 Locks in Series

A crucial property of scheduling a single lock is the relationship between arrival times of ships and the starting times of the lockings. A locking will only start at an integer time difference of the arrival time of a ship. A similar property was already used in Lemma 7, where R_c was the set of possible equivalent release times.

Baptiste (1999) defines this property and uses it as the basis of a dynamic programming algorithm:

Lemma 10 *Let*

$$\Theta = \{t = r_i + l * p | l \in \mathbb{N}\}. \tag{15}$$

Every task will start at a time $s_i \in \Theta$.

This property also holds for the SLSP1 and can be proven in a similar way as Baptiste (1999).

Regardless of the applications in dynamic programming this property can be used in discretisations of a given model. Assume that multiple locks are put in series, with a canal of given length in between, then the arrival time of a ship r_i will not be a constant anymore, but will depend on the starting time at the previous lock.

Consider the following simple example with 2 locks P and Q. The following holds:

$$\Theta_P = \left\{r_i^P + lp\right\} \cup \{\theta_Q + \Delta T + lp, \theta_Q \in \Theta_Q\} \tag{16}$$

$$\Theta_Q = \left\{r_i^Q + lp\right\} \cup \{\theta_P + \Delta T + lp, \theta_P \in \Theta_P\} \tag{17}$$

with r_i^P the arrival times of ships that start at lock P (without going past Q) and r_i^Q these of ships starting at lock Q. ΔT is the travelling time between the locks and $l \in \mathbb{N}$.

Filling in and repeating the substitution of Θ_Q in Θ_P (θ_P) yields

$$\Theta_P = \left\{r_i^P + 2k\Delta t + lp\right\} \cup \left\{r_i^Q + (2k+1)\Delta t + lp\right\}$$

(18)

In this case a single degree of freedom k is added.

The same reasoning can also be applied to 3 locks P, Q and R. After a similar deduction, the following holds for the middle lock Q:

$$\Theta_Q = \begin{array}{l} \left\{r_i^P + (2k+1)\Delta T_P + (2m)\Delta T_Q + lp\right\} \\ \quad \cup \left\{r_i^Q + (2k+1)\Delta T_Q + (2m)\Delta t_P + lp\right\} \end{array} \tag{19}$$

with ΔT_P the travelling time between P and Q and ΔT_Q the time between Q and R. In this case two degrees of freedom, k and m have been added. This increase in degrees of freedom lets the possible values in Θ rise exponentially, which makes the usage of Lemma 10 as the basis for an algorithm for multiple locks in series hard.

Although the complexity seems to rise with the number of locks in series, experimental results point out that the further the locks are apart (higher ΔT), the less influence they will have on each other as shown by Ting and Schonfeld (1998). Their results are obtained by comparing a heuristic algorithm for a single lock with an integrated algorithm for multiple locks, so they should not be generalized. Based on the formulas above, it is however clear that if a fixed time horizon T is used (such that $\forall\, \theta \in \Theta : \theta < T$), the number of elements in Θ will decrease with increasing ΔT.

6 Conclusions

In this paper we have shown that the SLOPminimax is NP-hard, since fitting the ships inside the lock is a packing problem. We have however shown that the bidirectional, alternating traffic through a lock is no problem and a polynomial time algorithm exists for the SLOP1. A practical algorithm with time complexity $O(n^4 \log n)$ and memory complexity $O(n^2)$ is presented.

Acknowledgements

I would like to thank my supervisor Karl Meerbergen and mentor Luk Knapen for their valuable advise when working on my Master's thesis and for giving me the trust and freedom to explore this scheduling problem in such detail. Special thanks go out to Luk Knapen for introducing me into the world of inland shipping and sharing his many years of experience in lock scheduling. Thanks also to my PhD promotors, Bart Preneel and Frederik Vercauteren, for allowing me to continue work on this scheduling problem at COSIC. Finally I would like to thank the anonymous reviewers for taking the time to dig into this paper. I've hardly ever seen such extensive and in-depth comments which truly helped improving the paper.

References

Baptiste P (1999) Polynomial time Algorithms for Minimizing the Weighted Number of Late Jobs on a Single Machine with Equal Processing Times. Journal of Scheduling 2(6):245–252

Baptiste P (2000) Scheduling Equal-Length Jobs on Identical Parallel Machines. Discrete Applied Mathematics 103(1-3):21–32

Brucker P, Knust S (2009) Complexity Results for Scheduling Problems. URL `http://www.informatik.uni-osnabrueck.de/knust/class/`

Chrobak M, Dürr C, Jawor W, Kowalik L, Kurowski M (2006) A Note on Scheduling Equal-Length Jobs to Maximize Throughput. J Scheduling 9(1):71–73

FKS (2007) Formal & Knowledge Systems - Waterborne. `http://www.fks-waterborne.eu/`

Garey M, Johnson D, Simons B, Tarjan R (1981) Scheduling Unit–Time Tasks with Arbitrary Release Times and Deadlines. SIAM Journal on Computing 10(2):256–269

Graham R, Lawler E, Lenstra J, Kan A (1979) Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. Annals of Discrete Mathematics 5(2):287–326

Hermans J (2008) Optimalisatie van binnenscheepvaart. Master's thesis, KU Leuven

Lawler E (1973) Optimal Sequencing of a Single Machine Subject to Precedence Constraints. Management Science 19(5):544–546

Lodi A, Martello S, Vigo D (2002) Recent Advances on Two-Dimensional Bin Packing Problems. Discrete Applied Mathematics 123(1-3):379–396

López-Ortiz A, Quimper CG (2011) A Fast Algorithm for Multi-Machine Scheduling Problems with Jobs of Equal Processing Times. In: Schwentick T, Dürr C (eds) STACS, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, LIPIcs, vol 9, pp 380–391

Mundy R, Campbell J (2005) Management Systems for Inland Waterway Traffic Control. Tech. rep., `http://www.ctre.iastate.edu/mtc/reports/inland_waterway/volume1.htm`

Simons B (1983) Multiprocessor Scheduling of Unit-Time Jobs with Arbitrary Release Times and Deadlines. SIAM Journal on Computing 12:294

Ting C, Schonfeld P (1998) Integrated Control for Series of Waterway Locks. Journal of Waterway, Port, Coastal and Ocean Engineering 124(4):199–207

Zhang X, Qi H, Fu X, Yuan X (2007) Hybrid Algorithm to Minimize Total Weighted Wait-Time of Ships for Navigation Co-Scheduling in the Three Gorges Project. International Conference on Transportation Engineering pp 2759–2764

A Additional Code Listings

Algorithm 4 Critical time subroutine (Backscheduling algorithm)

for T_j with $d_j \geq d_i$ **do**
  $C \leftarrow \infty$
  if $I_{i+1,j}$ is not defined **then**
    Set $I_{i,j} \leftarrow (d_j - 1, d_j, 1, 0)$ if $o_i = A$ and $I_{i,j} \leftarrow (d_j, d_j - 1, 0, 1)$ if $o_i = B$.
  else if $I_{i,j}$ is not defined **then**
    if $o_i = A$ **then**
      $\#A_{i,j} \leftarrow \#A_{i+1,j} + 1$, $\#B_{i,j} \leftarrow \#B_{i+1,j}$
      $LA \leftarrow \max(\#B_{i+1,j} - \#A_{i+1,j} - 1, 0)$
      $LB \leftarrow \max(\#A_{i+1,j} - \#B_{i+1,j} - 1, 0)$
      $c^A_{i,j} \leftarrow c^A_{i+1,j}$
      if $LA = 0$ **then**
        if $\#A_{i+1,j} \leq \#B_{i+1,j}$ **then**
          $c^A_{i,j} \leftarrow c^A_{i,j} - 1$
          If $\exists FB_k$ s.t. $c^A_{i,j}$, then $c^A_{i,j} \leftarrow \inf(FB_k)$.
        end if
        $c^A_{i,j} \leftarrow c^A_{i,j} - 1$
        If $\exists FA_k$ s.t. $c^A_{i,j}$, then $c^A_{i,j} \leftarrow \inf(FA_k)$.
      end if
      (Analogous for the computation of $c^B_{i,j}$).
      $I_{i,j} \leftarrow (c^A_{i,j}, c^B_{i,j}, \#A_{i,j}, \#B_{i,j})$.
    else
      (Analogous to the case $o_i = A$.)
    end if
  end if
  if $\#X_{i,j} > \#\bar{X}_{i,j}$ (with X as loaded from R in the outer loop) **then**
    $C \leftarrow \min(C, c^X_{i,j})$
  end if
end for
Return C